Curso de Optimización I (DEMAT/CIMAT)

Tarea 2

Descripción:	Fechas
Fecha de publicación del documento:	**Febrero 4, 2024**
Fecha límite de entrega de la tarea:	**Febrero 11, 2024**

Indicaciones

Puede escribir el código de los algoritmos que se piden en una celda de este notebook o si lo prefiere, escribir las funciones en un archivo `.py` independiente e importar la funciones para usarlas en este notebook. Lo importante es que en el notebook aparezcan los resultados de la pruebas realizadas y que:

- Si se requieren otros archivos para poder reproducir los resultados, para mandar la tarea cree un archivo ZIP en el que incluya el notebook y los archivos adicionales.
- Si todos los códigos para que se requieren para reproducir los resultados están en el notebook, no hace falta comprimir el notebook y puede anexar este archivo en la tarea del Classroom.
- Exportar el notebook a un archivo PDF y anexarlo en la tarea del Classroom como un archivo independiente. **No incluya el PDF dentro del ZIP**, porque la idea que lo pueda accesar directamente para poner anotaciones y la calificación de cada ejercicio.

En la descripción de los ejercicios se nombran algunas variables para el algoritmo, pero sólo es para facilitar la descripción. En la implementación pueden nombrar sus variables como gusten.

En los algoritmos se describen las entradas de las funciones. La intención es que tomen en cuenta lo que requiere el algoritmo y que tiene que haber parámetros que permitan controlar el comportamiento del algoritmo, evitando que dejen fijo un valor y que no se puede modificar para hacer diferentes pruebas. Si quieren dar esta información usando un tipo de dato que contenga todos los valores o usar variables por separado, etc., lo pueden hacer y no usen variables globales si no es necesario.

Lo mismo para los valores que devuelve una función. Pueden codificar como gusten la manera en que regresa los cálculos. El punto es que podamos tener acceso a los resultados para poder usarlos, y por eso no es conveniente que la función sólo imprima los valores sin devolverlos.

Para los ejercicios teóricos puede escribir en la celda la solución, o si escribio la solución en una hoja, puede insertar una(s) foto(s) en la que se vea clara la solución. Si le es más facíl insertar la imagen en un procesador de texto como Word, lo puede utilizar y exportar el documento a PDF y subir el archivo. No lo compacte para que se pueda escribir anotaciones en el PDF.

Ejercicio 1 (2 puntos)

Estimar la cantidad de iteraciones que requiere el algoritmo de descenso máximo con paso exacto para alcanzar el minimizador $\mathbf{x}_*$ de la función cuadrática

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$$

donde $\mathbf{A}$ es una matriz simétrica y definida positiva que su información y la del arreglo $\mathbf{b}$ está almacenada en archivo en formato `npz` contenido en el archivo `datosTarea02.zip`.

Para hacer esto, calculamos el minimizador $\mathbf{x}_*$ de $f(\mathbf{x})$ resolviendo el sistema de ecuaciones $\mathbf{A}\mathbf{x}_* = \mathbf{b}$ y definimos

$$q(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}_*)^\top \mathbf{A}(\mathbf{x} - \mathbf{x}_*).$$

Sabemos que $q(\mathbf{x})$ y $f(\mathbf{x})$ sólo difieren en una constante y podemos usar $q(\mathbf{x})$ para estimar la manera en que decrece la función mediante el resultado de la Proposición 6 de la Clase 6:

$$q(\mathbf{x}_{k+1}) \leq \left(\frac{\lambda_{\max}(\mathbf{A}) - \lambda_{\min}(\mathbf{A})}{\lambda_{\max}(\mathbf{A}) + \lambda_{\min}(\mathbf{A})}\right)^2 q(\mathbf{x}_k).$$

Si

$$c = \frac{\lambda_{\max}(\mathbf{A}) - \lambda_{\min}(\mathbf{A})}{\lambda_{\max}(\mathbf{A}) + \lambda_{\min}(\mathbf{A})},$$

entonces

$$q(\mathbf{x}_{k+1}) \leq c^2 q(\mathbf{x}_k) \leq c^4 q(\mathbf{x}_{k-1}) \leq c^6 q(\mathbf{x}_{k-2}) \leq \ldots \leq c^{2k} q(\mathbf{x}_1) \leq c^{2(k+1)} q(\mathbf{x}_0).$$

Como

$$2q(\mathbf{x}_k) = (\mathbf{x} - \mathbf{x}_*)^\top \mathbf{A}(\mathbf{x} - \mathbf{x}_*) = \|\mathbf{x} - \mathbf{x}_*\|_{\mathbf{A}}^2,$$

$q(\mathbf{x}_k)$ es una medida de la distancia al cuadrado de $\mathbf{x}_k$ a $\mathbf{x}_*$, de modo dada una tolerancia $\tau > 0$ podemos buscar el valor k para el cual se cumpla

$$\|\mathbf{x}_k - \mathbf{x}_*\|_{\mathbf{A}} = \sqrt{2q(\mathbf{x}_k)} \leq c^k \sqrt{2q(\mathbf{x}_0)} < \tau$$

y ese k es la estimación de la cantidad de iteraciones que requiere el algoritmo.

1. Escriba una función que reciba como parámetro el nombre de un archivo `npz`, lea el archivo y cree la matriz $\mathbf{A}$ y el vector $\mathbf{b}$ del archivo `npz`, y calcule el minimizador $\mathbf{x}_*$ de $f(\mathbf{x})$ resolviendo el sistema de ecuaciones $\mathbf{A}\mathbf{x}_* = \mathbf{b}$. Use la factorización de Cholesky para resolver el sistema de ecuaciones y de esta manera saber si la matriz es definida positiva, y en este caso devolver $\mathbf{A}$, $\mathbf{b}$ y $\mathbf{x}_*$. En caso contrario devolver $\mathbf{A}$, $\mathbf{b}$ y $None$.

2. Programe la función que evalua la función $q(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}_*)^\top \mathbf{A}(\mathbf{x} - \mathbf{x}_*)$. La función recibe como parámetros el punto $\mathbf{x}$, la matriz $\mathbf{A}$ y el punto $\mathbf{x}_*$ y devolver el valor de $q(\mathbf{x})$.

3. Programe una función estima la cantidad de iteraciones que el algoritmo requiere. Esta función recibe como argumentos la matriz $\mathbf{A}$, el punto $\mathbf{x}_0$, el punto $\mathbf{x}_*$ y una tolerancia $\tau > 0$. La función calcula la

cantidad c descrita anteriormente y determina el entero k que cumple con $c^k\sqrt{2q(\mathbf{x}_0)} < \tau$. La función debe devolver k y c.

4. Pruebe la función del punto anterior usando los datos de cada archivo `npz` contenidos en el archivo `datosTarea02.zip`. Use la función del Punto 1 y si se pudo calcular $\mathbf{x}_*$, defina n como el tamaño del vector $\mathbf{b}$, el punto inicial $\mathbf{x}_0 = (10, 10, \ldots, 10)^\top$ de dimensión n y
ejecute la función del Punto 3 usando como tolerancia $\tau = \sqrt{\epsilon_m}$, donde ϵ_m es el épsilon de la máquina.

   Imprima el valor n, $q(\mathbf{x}_0)$, k, c.

Nota: Cada archivo `npz` en el ZIP tiene dos arreglos que corresponden a la matriz $\mathbf{A}$ y el vector $\mathbf{b}$. Para leer los datos puede hacer, por ejemplo:

```
npzfile = np.load("datosTarea02/matA_vecb1.npz")
A  = npzfile['arr_0']
b  = npzfile['arr_1']
```

Solución:

In [1]:
```
import numpy as np
from scipy import linalg

def read_Ab(filepath):
    """Lee la matriz A y el vector b del archivo """
    npzfile = np.load(filepath)
    A = npzfile['arr_0']
    b = npzfile['arr_1']
    return A, b

def solve_cholesky(A: np.ndarray, b:np.ndarray):
    """Resuelve el sistema Ax=b mediante la factorizacion de Cholesky
    (y de paso asegurandose que A es simetrica y positiva definida)"""
    try:
        L = np.linalg.cholesky(A)
        x = linalg.cho_solve((L, True), b)
        return A, b, x
    except np.linalg.LinAlgError:
        print("La matriz no es simetrica o definida positiva")
        return A, b, None

def opti_of_f_Ab(filepath):
    """Determina A, b y x*"""
    A, b = read_Ab(filepath)
    return solve_cholesky(A, b)

def q(x: np.ndarray, A: np.ndarray, xs: np.ndarray):
    return 1/2*(x-xs).T@A@(x-xs)

def estimate_iter(A: np.ndarray, x0, xs, t):
    try:
        eigv = np.linalg.eigvalsh(A) # Pues A es simetrica
    except np.linalg.LinAlgError:
        print("El calculo del valor propio no convergio.")
```

```
    else:
        c = (eigv[-1]- eigv[0])/(eigv[-1] + eigv[0])
        bound = c*np.sqrt(2*q(x0, A, xs))
        k = 1
        while bound>t:
            bound *= c
            k += 1
        return k, c
```

In [2]:
```
# filepath = "/content/datosTarea02/matA_vecb.npz"

files = [f'/content/datosTarea02/matA_vecb{i}.npz' for i in range(1,5)]
for file in files:
    A, b, x = opti_of_f_Ab(file)
    if x is not None:
        n = b.shape[0]
        x0 = np.repeat(10, n)
        t = np.sqrt(np.finfo(float).eps)
        k, c = estimate_iter(A, x0, x, t)
        print(f"n={n}, q(x0) = {q(x0, A, x)}, k={k}, c={c}")
```

```
n=2, q(x0) = 1113.15, k=54, c=0.6666666666666666
n=10, q(x0) = 2658.8249999999994, k=150, c=0.8610359125293412
n=100, q(x0) = 18134.27, k=8, c=0.041755326694693326
n=500, q(x0) = 543978.79, k=276, c=0.9132114471426375
```

_

Ejercicio 2 (3 puntos)

Programe el Algoritmo 2 de la Clase 5 para optimizar funciones cuadráticas de la forma

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$$

con el método de descenso máximo con paso exacto.

1. La función que implementa el algoritmo recibe como argumentos:

- la matriz $\mathbf{A}$ (que se supone que es simétrica y definida positiva),
- el vector $\mathbf{b}$ de la función cuadrática,
- un punto inicial $\mathbf{x}_0$
- una tolerancia τ y
- el número máximo de iteraciones N.

La función debe devolver:

- El último punto $\mathbf{x}_k$ generado por el algoritmo,
- el número k de iteraciones realizadas y
- Una variable indicadora que es $True$ si el algoritmo termina por cumplirse la condición de paro ($\|\alpha_k \mathbf{g}_k\| < \tau$) o $False$ si termina porque se alcanzó el número máximo de iteraciones.

2. Programe la función que evalúa la función $f(\mathbf{x})$. La función recibe como argumentos la matriz $\mathbf{A}$ y el vector $\mathbf{b}$, y devuelve el valor $\frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} - \mathbf{b}^\top \mathbf{x}$.

3. Pruebe el algoritmo con las matrices y vectores que se encuentran en los archivos `npz` que están contenidos en el archivo `datosTarea02.zip` :

Para cada archivo npy haga lo siguiente:

- Use la función del Punto 1 del Ejercicio 1 para obtener $\mathbf{A}$, $\mathbf{b}$ y $\mathbf{x}_*$. Si $\mathbf{x}_*$ no es `None` continúe y defina la variable n como el tamaño del vector $\mathbf{b}$. Imprima el valor de n para saber la dimensión de la variable $\mathbf{x}$.
- Haga $\mathbf{x}_0 = (10, 10, \ldots, 10)^\top$ de dimensión n.
- Defina la tolerancia $\tau = \sqrt{\epsilon_m}$, donde ϵ_m es el épsilon de la máquina.
- Calcule el punto $\mathbf{x}_k$ con el algoritmo. Elija el número de iteraciones máximas para el algoritmo. Puede tomar como referencia el resultado en el Ejercicio 1.
- Imprima los valores

$$f(\mathbf{x}_0), k, f(\mathbf{x}_k), \|\mathbf{x}_k - \mathbf{x}_*\|,$$

y $\mathbf{x}_k$ si $n \leq 6$, o los primeros tres elementos y los últimos tres elementos del arreglo $\mathbf{x}_k$ si $n > 6$.

4. Escriba un comentario sobre si el número de iteraciones estimadas fue una buena cota superior.

Solución:

In [3]:
```
import numpy as np

def max_quadratic_descent(A: np.ndarray, b: np.ndarray, x0: np.ndarray, t, N):
    for k in range(N):
        gk = A@x0 - b
        ak = (gk.T@gk)/(gk.T@A@gk)
        if np.linalg.norm(ak*gk) <t:
            return x0, k, True
        x0 = x0 - ak*gk
    return x0, N, False

def f(x, A, b):
    return (x.T@A@x - b.T@x)/2
```

In [4]:
```
def test_max_quadratic_descent(file):
    A, b, x = opti_of_f_Ab(file)
    if x is not None:
        n = b.shape[0]
        x0 = np.repeat(10, n)
        t = np.sqrt(np.finfo(float).eps)
        N, _ = estimate_iter(A, x0, x, t)
```

```
        xk, k, _ = max_quadratic_descent(A, b, x0, t, N)
        print(f"f(x0)={f(x0, A, b)}, k={k}, f(xk) = {f(xk, A, b)}, ||xk-x*|| =  {np.linalg.norm(
```

In [5]:
```
for file in files:
    test_max_quadratic_descent(file)
```

```
f(x0)=1055.0, k=5, f(xk) = -5.111107093114242e-10, ||xk-x*|| =  1.4608712423941497e-09
f(x0)=2954.25, k=105, f(xk) = -5.446987216828347e-08, ||xk-x*|| =  6.309047416687804e-08
f(x0)=16485.7, k=7, f(xk) = 8.042150057008257e-09, ||xk-x*|| =  1.2326882730502574e-09
f(x0)=543535.8, k=117, f(xk) = -2.4260037889689556e-07, ||xk-x*|| =  1.0879857374046182e-07
```

4

En general me parece que el estimador es bueno como una cota superior al numero de interaciones, pues hubo casos donde si era "muy" grande en comparación pero otro donde estaba basicamente igual.

_

Ejercicio 3 (3.5 puntos)

Programe el Algoritmo 1 de la Clase 5 de descenso máximo, usando el método de la sección dorada para obtener $\alpha_k \in [0, 1]$:

$$\alpha_k = \arg \min_{\alpha \in [0,1]} f(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k)).$$

1. La función que implementa el algoritmo recibe como entrada:

- La función $f(\mathbf{x})$,
- el gradiente $\nabla f(\mathbf{x})$ de la función f,
- un punto inicial $\mathbf{x}_0$,
- las tolerancia $\tau_1 > 0$ y $\tau_2 > 0$,
- el número máximo de iteraciones N para el algoritmo de descenso máximo, y
- el número máximo de iteraciones N_{gs} para el método de la sección dorada.

La función devuelve

- El último punto $\mathbf{x}_k$ generado por el algoritmo,
- el número k de iteraciones realizadas y
- Una variable indicadora que es $True$ si el algoritmo termina por cumplirse la condición de paro ($\|\alpha_k \mathbf{p}_k\| < \tau_1$) o $False$ si termina porque se alcanzó el número máximo de iteraciones.

- Un arreglo que contiene la secuencia de puntos $\mathbf{x}_0, \mathbf{x}_1, \dots \mathbf{x}_k$ si la dimensión de la variable es $n = 2$, y es vacío en otro caso. Es decir, sólo cuando la dimensión guardamos la secuencia de puntos.

Dentro de esta función se puede definir $\phi(\alpha) = f(\mathbf{x}_k - \alpha \nabla f(\mathbf{x}_k))$ y usar el algoritmo de la sección dorada de la Tarea 1 para calcular su minimizador α_k en el intervalo $[0, 1]$. Para hacer esto, puede usar una función `lambda` como en el ejemplo en las notas de la **Ayudantía 2**. Use la tolerancia τ_2 para el algoritmo de la sección dorada.

2. Para probar el algoritmo, programe las siguientes funciones, calcule su gradiente de manera analítica y programe la función correspondiente. Use cada punto $\mathbf{x}_0$ como punto inicial del algoritmo.

Función de Himmelblau: Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2.$$

$$\mathbf{x}_0 = (2., 4.)$$

$$\mathbf{x}_0 = (0., 0.)$$

Función de Beale : Para $\mathbf{x} = (x_1, x_2)$

$$f(\mathbf{x}) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2.$$

$$\mathbf{x}_0 = (2., 3.)$$

$$\mathbf{x}_0 = (2., 4.)$$

Función de Rosenbrock: Para $\mathbf{x} = (x_1, x_2, \dots, x_n)$

$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \quad n \geq 2.$$

$$\mathbf{x}_0 = (-2.1, 4.5)$$

$$\mathbf{x}_0 = (-1.2, 1.0)$$

$$\mathbf{x}_0 = (-2.1, 4.5, -2.1, 4.5, -2.1, 4.5, -2.1, 4.5, -2.1, 4.5)$$

$$\mathbf{x}_0 = (-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0)$$

En la página Test functions for optimization pueden ver las gráficas de estas funciones y sus mínimos locales.

Use las tolerancias $\tau_1 = \sqrt{n}\epsilon_m^{1/3}, \tau_2 = \epsilon_m^{1/2}$, donde ϵ_m es el épsilon de la máquina, use el número de iteraciones máximas $N = 10000$ para el descenso máximo y $N_{gs} = 200$ para el método de la sección dorada.

Para las funciones de dos variables grafique los contornos de nivel. Modifique la función `contornosFnc2D`, o haga la suya, y pase como argumento la secuencia de puntos que devuelve el algoritmo para visualizar la trayectoria de los puntos $\mathbf{x}_k$.

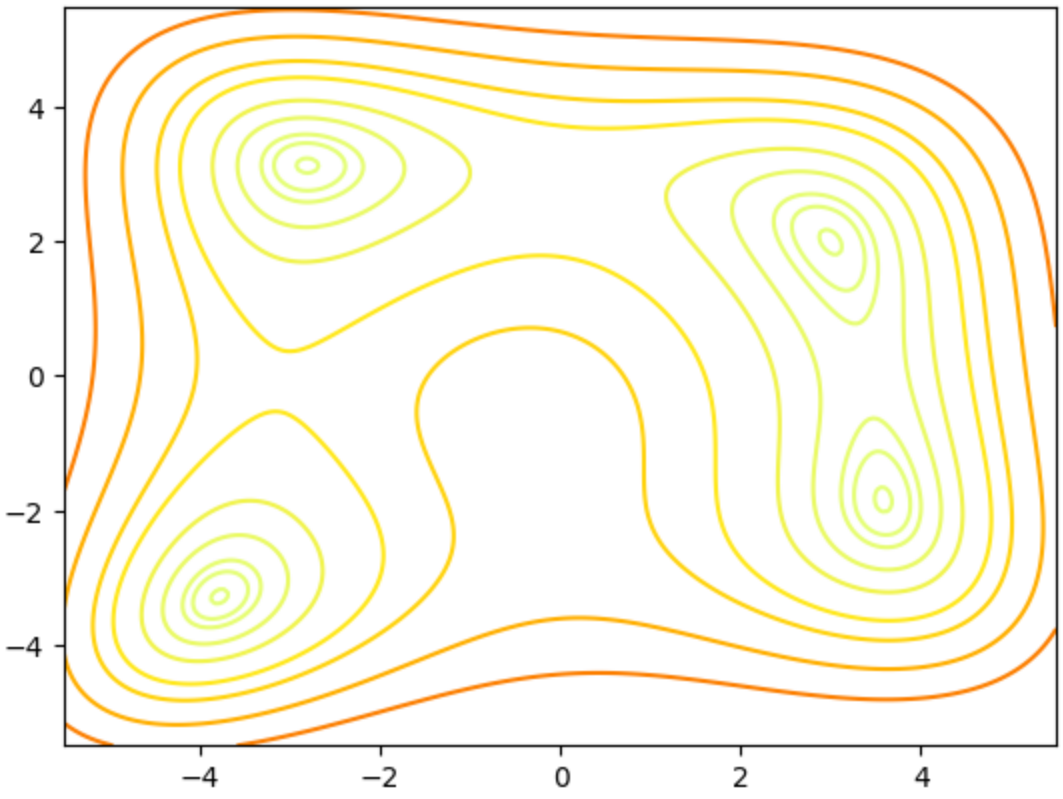
3. Repita la prueba para función de Rosenbrock usando el punto inicial $\mathbf{x}_0 = (-2.1, 4.5)$ usando $\tau_2 = \epsilon_m^{1/4}$ y $N_{gs} = 50$ para relajar las condiciones de paro del método de la sección dorada y ver si

podemos terminar más rápido. Escriba un comentario sobre si conviene hacer esto o cuando no conviene hacerlo.

Solución:

In [6]:
```
import matplotlib.pyplot as plt
import numpy as np

def fncHimmelblau(x, fparam=None):
    return (x[0]**2 + x[1] - 11)**2 + (x[0] + x[1]**2 - 7)**2;


def contornosFnc2D(fncf, xleft, xright, ybottom, ytop, levels):
    # Crea una discretización uniforme del intervalo [xleft, xright]
    ax = np.linspace(xleft, xright, 250)
    # Crea una discretización uniforme del intervalo [ybottom, ytop]
    ay = np.linspace(ybottom, ytop, 200)
    # La matriz mX que tiene las abscisas
    mX, mY = np.meshgrid(ax, ay)
    # Se crea el arreglo mZ con los valores de la función en cada nodo
    mZ = mX.copy()
    for i,y in enumerate(ay):
        for j,x in enumerate(ax):
            mZ[i,j] = fncf(np.array([x,y]))
    # Grafica de las curvas de nivel
    fig, ax = plt.subplots()
    CS = ax.contour(mX, mY, mZ, levels, cmap='Wistia')
    return fig, ax

contornosFnc2D(fncHimmelblau, xleft=-5.5, xright=5.5, ybottom=-5.5, ytop=5.5,
               levels=[0.5, 5, 10, 25, 50, 100, 150, 250, 400])
```

Out[6]:
```
(<Figure size 640x480 with 1 Axes>, <Axes: >)
```



In [7]:
```
import numpy as np
from scipy.optimize import golden

eps = np.finfo(float).eps

def max_descent(f, gf, x0, t1, t2, N, Ngs):
    steps = []
    for k in range(N):
        steps.append(x0)
        gk = gf(x0)
        pk = -gk
        ak = golden(lambda t: f(x0 + t*pk), brack=(0, 1), tol = t2, maxiter=Ngs)
        if np.linalg.norm(ak*pk) < t1:
            return x0, k, True, steps
        x0 = x0 + ak*pk
    return x0, N, False, steps

def test_max_descent(f, gf, initpoints, t2=np.sqrt(eps), N=10**4, Ngs=200):
    # setup
    fig, ax = contornosFnc2D(f, xleft=-5.5, xright=5.5, ybottom=-5.5, ytop=5.5,
               levels=[0.5, 5, 10, 25, 50, 100, 150, 250, 400])
    for point in initpoints:
        t1 = np.sqrt(len(point))*eps**(1/3)
        x0, N, _, steps = max_descent(f, gf, point,  t1, t2, N, Ngs)
        print(x0, N)
        ax.plot(x0[0], x0[1],'ro', )
        if len(x0) ==2:

            ax.plot(*np.array(steps).T)
```

In [7]:
```
```

Para la función de Himmelblau

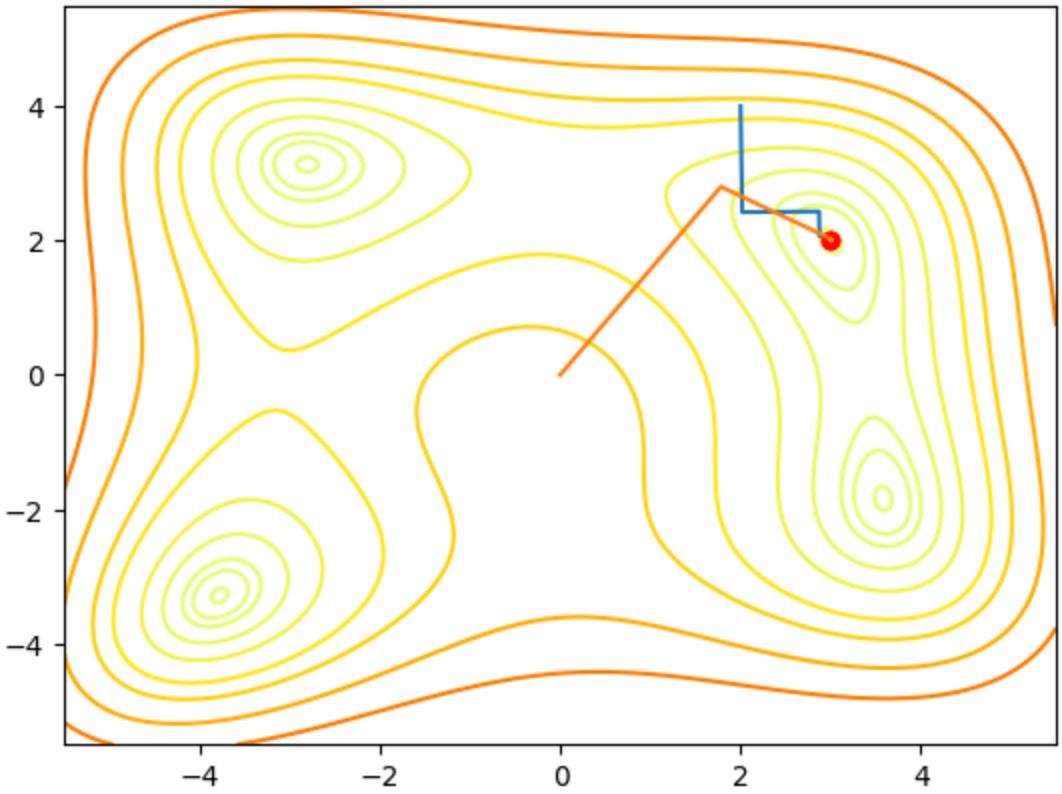
Tenemos que

$$\nabla f(\mathbf{x}) = \begin{pmatrix} 4x_1(x_1^2 + x_2 - 11) + 2(x_1 + x_2^2 - 7) \\ 2(x_1^2 + x_2 - 11) + 4x_2(x_1 + x_2^2 - 7) \end{pmatrix}$$

In [8]:
```
# gradiente
def grdHimmelblau(x):
    x, y = x[0], x[1]
    x1 = 4*x*(x**2 + y - 11) + 2*(x + y**2 - 7)
    x2 = 2*(x**2 + y - 11) + 4*y*(x + y**2 - 7)
    return np.array([x1, x2])
```

In [9]:
```
# fig, ax = contornosFnc2D(fncHimmelblau, xleft=-5.5, xright=5.5, ybottom=-5.5, ytop=5.5,
#                levels=[0.5, 5, 10, 25, 50, 100, 150, 250, 400])
# ax.plot(x0[0], x0[1],'ro', )
points = [[2, 4], [0, 0]]
test_max_descent(fncHimmelblau, grdHimmelblau, points)
```

```
[2.99998996 2.00000569] 13
[2.99999201 2.00001148] 13
```



Para Beale

Tenemos que

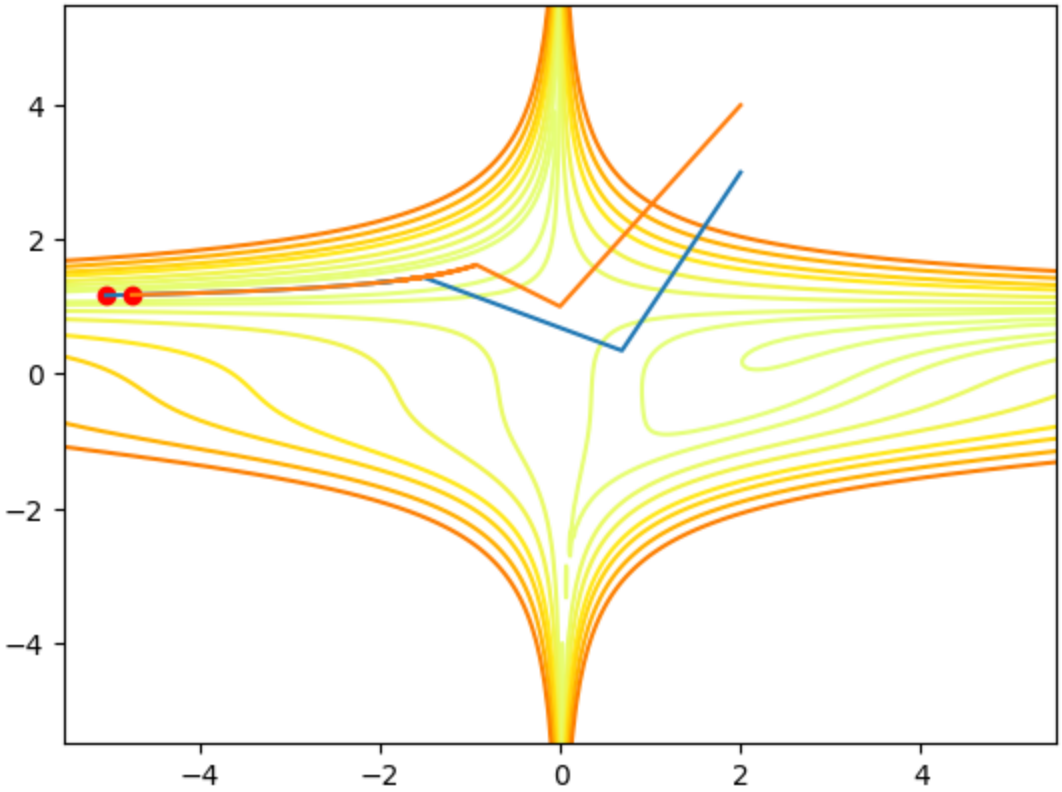
$$\nabla f(\mathbf{x}) = \begin{pmatrix} 2(1.5 - x_1 + x_1x_2)(x_2 - 1) + 2(2.25 - x_1 + x_1x_2^2)(x_2^2 - 1) + 2(2.625 - x_1 + x_1x_2^3)(x_2^3 - 1) \\ 2(1.5 - x_1 + x_1x_2)x_1 + 4(2.25 - x_1 + x_1x_2^2)x_1x_2 + 6(2.625 - x_1 + x_1x_2^3)x_1x_2^2 \end{pmatrix}$$

In [10]:
```python
import numpy as np

def Beale(x:np.ndarray):
    a, b = x[0], x[1]
    p = (1.5 - a + a*b)**2
    q = (2.25 - a + a*b**2)**2
    r = (2.625 -a + a*b**3)**2
    return p + q + r
# def beale2(x: np.ndarray):


def grdBeale(x):
    a, b = x[0], x[1]
    x1 = 2*(1.5 - a +a*b)*(b-1) + 2*(2.25 - a + a*b**2)*(b**2-1)
    x1 += 2*(2.625 - a + a*b**3)*(b**3 - 1)
    x2 = 2*(1.5 - a + a*b)*a + 4*(2.25 -a + a*b**2)*a*b
    x2 += 6*(2.625 - a + a*b**3)*a*b**2
    return np.array([x1, x2])
```

In [11]:
```python
points = [[2, 3], [2, 4]]
```

```
test_max_descent(Beale, grdBeale, points)
```

```
[-5.03736277  1.16919966] 10000
[-4.75064655  1.17800261] 10000
```



Para Rosenbrock

Se tiene que

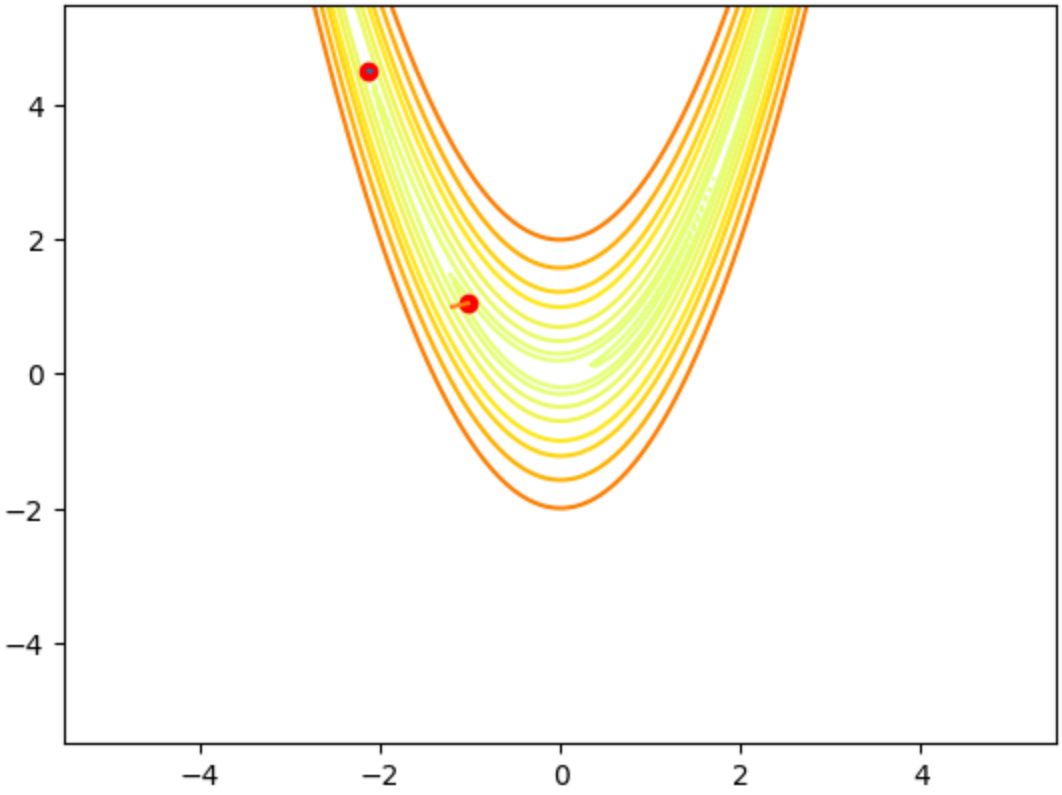
$$\nabla f(\mathbf{x}) = \begin{pmatrix} -4(x_2 - x_1^2) - 1 \\ 2(x_2 - x_1^2) + -4(x_3 - x_2^2) - 1 \\ \vdots \\ 2(x_n - {x_{n-1}}^2) \end{pmatrix}$$

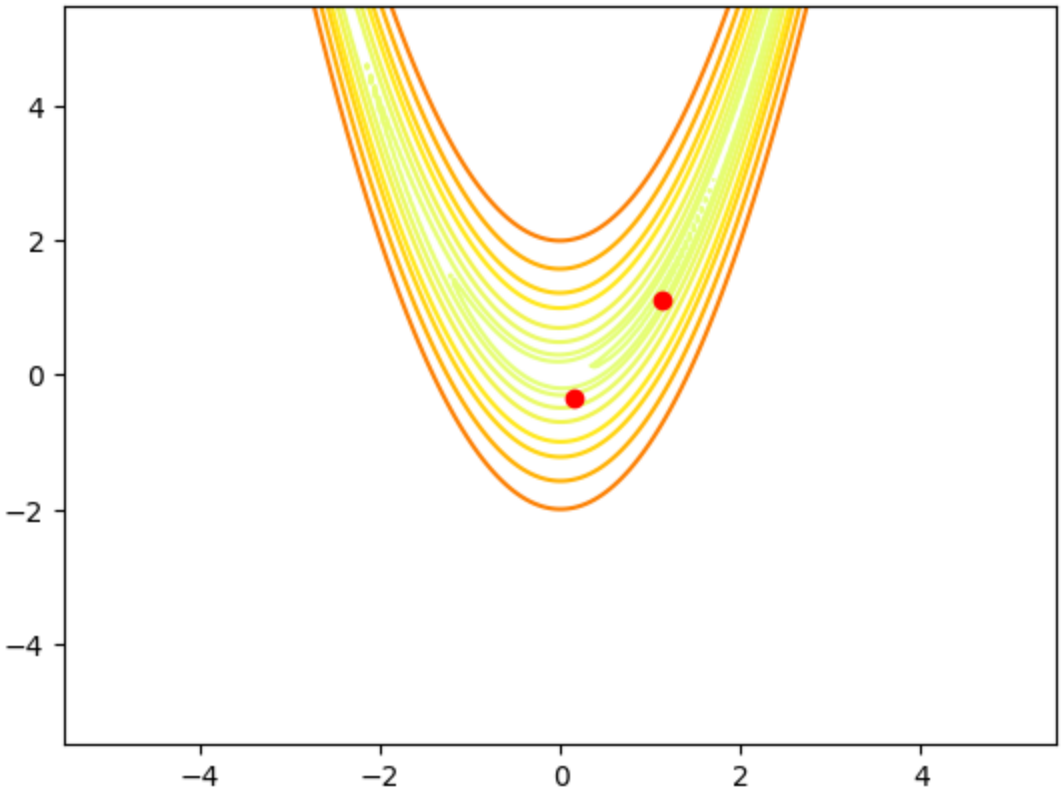
In [12]:
```
def rossenbrock(x: np.ndarray):
    n = x.shape[0]

    y = 0
    for i in range(n-1):
        y += 100*(x[i+1] -x[i]**2)**2 + (1-x[i])**2
    return y

def grdRossenbrock(x: np.ndarray):
    n = x.shape[0]
    grad = [-4*(x[1] - x[0]**2)*x[0] -1]
    # x1 = -4*(x[1] - x[0]**2)*x[0] -1
    for i in range(1, n-1):
        y = 2*(x[i] - x[i-1]**2)
        y += -4*(x[i+1] - x[i]**2) - 1
        grad.append(y)
```

```
    grad.append(2*(x[-1] - x[-2]**2))
    return np.array(grad)
```

In [13]:
```
points = np.array([[-2.1, 4.5], [-1.2, 1.0]])
test_max_descent(rossenbrock, grdRossenbrock, points)
points = np.array([[-2.1, 4.5, -2.1, 4.5, -2.1, 4.5, -2.1, 4.5, -2.1, 4.5],
 [-1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0, -1.2, 1.0]])
test_max_descent(rossenbrock, grdRossenbrock, points)
```

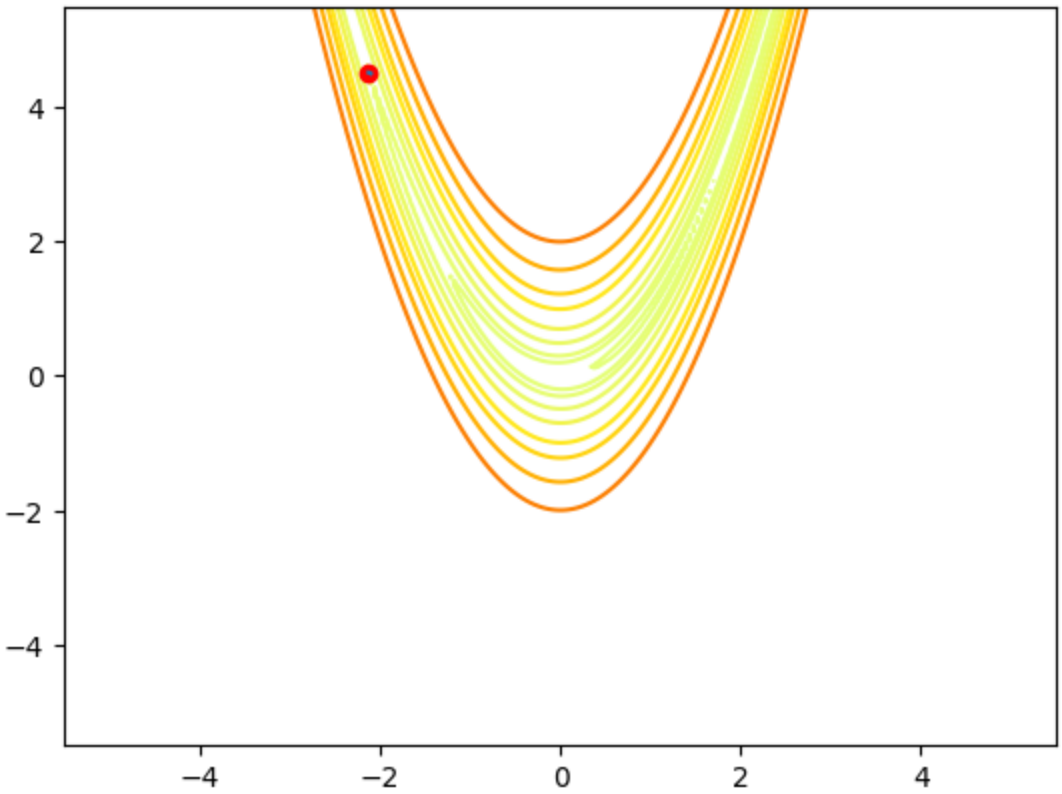
```
[-2.123625    4.51717113] 2
[-1.02013669  1.05066844] 2
[1.13410153 1.11573373 1.05876569 0.98401217 0.95552737 1.02253305
 1.17920245 1.42753932 1.93524813 3.52235445] 9
[ 0.14606835 -0.33774697 -0.51904999 -0.27545612 -0.51439293 -0.27230286
 -0.36957023  0.59465479  0.81632923  0.66956607] 9
```





In [14]:
```
x0 = np.array([-2.1, 4.5])
test_max_descent(rossenbrock, grdRossenbrock, [x0], t2=eps*(1/4), Ngs=50)
```

```
[-2.123625    4.51717113] 2
```



En mi caso parecce que no hubo gran mejora

_

Ejercicio 4 (1.5 puntos)

Sea $f(x) = (x-1)^2$ con $x \in \mathbb{R}$ y generamos la secuencia

$$x_{k+1} = x_k - \frac{\alpha}{2^k} f'(x_k)$$

con $0 < \alpha < 1$, para obtener el minimizador de la función $f(x)$.

¿Tiene este algoritmo la propiedad de descenso, es decir, $f(x_{k+1}) < f(x_k)$ a partir de un cierto k? ¿Es el algoritmo globalmente convergente?

Solución:

Primero notemos que $f'(x) = 2(x-1)$, por lo cual se cumple que

$$\begin{aligned} x_{k+1} &= x_k - \frac{\alpha}{2^k} f'(x_k) \\ &= x_k - \frac{\alpha}{2^k} 2(x_k - 1) \\ &= x_k - \frac{\alpha}{2^{k-1}} x_k + \frac{\alpha}{2^{k-1}} \\ &= x_k \left(1 - \frac{\alpha}{2^{k-1}}\right) + \frac{\alpha}{2^{k-1}}, \end{aligned}$$

de lo anterior tenemos que

$$\begin{aligned} |x_{k+1} - 1| &= \left| x_k \left(1 - \frac{\alpha}{2^{k-1}}\right) + \frac{\alpha}{2^{k-1}} - 1 \right| \\ &= \left| x_k \left(1 - \frac{\alpha}{2^{k-1}}\right) - \left(1 - \frac{\alpha}{2^{k-1}}\right) \right| \\ &= \left| \left(1 - \frac{\alpha}{2^{k-1}}\right) (x_k - 1) \right| \\ &= \left| \left(1 - \frac{\alpha}{2^{k-1}}\right) \right| |x_k - 1| . \end{aligned}$$

Un argumento inductivo (o recursivo?? ni idea) nos muestra que

$$|x_{k+1} - 1| = |x_0 - 1| \prod_{i=0}^{k} \left| \left(1 - \frac{\alpha}{2^{k-1}}\right) \right| .$$

Entonces se cumple que $x_k \to 1$ si y solo si $\prod_{i=0}^{k} \left|\left(1 - \frac{\alpha}{2^{k-1}}\right)\right| \to 0$. Notemos que para $i \geq 1$ se cumple que $0 < \frac{\alpha}{2^{k-1}} < 1$, pues $\alpha \in (0,1)$, luego $\prod_{i=1}^{k} \left|\left(1 - \frac{\alpha}{2^{k-1}}\right)\right| \to 0$ si si solo si $\sum_{i=1}^{k} \frac{\alpha}{2^{k-1}} \to \infty$, sin embargo sabemos que $\sum_{i=1}^{k} \frac{\alpha}{2^{k-1}} \to 2\alpha$, por lo cual la sucesión original no es convergente para $x_0 \neq 1$ o $\alpha \neq \frac{1}{2}$.

Ahora veamos si el algoritmo tiene la propiedad de decenso. Para ello notemos que

$$f(x_{k+1}) < f(x_k) \iff (x_{k+1} - 1)^2 < (x_k - 1)^2 \iff |x_{k+1} - 1| < |x_k - 1|,$$

por lo visto al inicio tenemos que

$$|x_{k+1} - 1| < |x_k - 1| \iff \left|\left(1 - \frac{\alpha}{2^{k-1}}\right)\right| |x_k - 1| < |x_k - 1| \iff \left|\left(1 - \frac{\alpha}{2^{k-1}}\right)\right| < 1,$$

como $\alpha \in (0,1)$ lo anterior es cierto y por tanto f tiene la propiedad de descenso.

In [14]: